

ECSE 426 - Microprocessor Systems

Final Project

GROUP 4

Matthew Lesko (260692352)

Marcel Morin

Fabrice Normandin

Harley Wiltzer (260690006)

April 22, 2018

Contents

1	Problem Statement	1
2	Theory	3
2.1	Bluetooth Low Energy	3
3	Implementation	5
3.1	An Overview of the System Architecture	5
3.2	The Beginning of the Pipeline: Handling the STM32 Discovery Board	6
3.2.1	Sending Microphone Data	7
3.2.2	Sending Accelerometer Data	8
3.3	Smart Phone Android Application	9
3.3.1	Brief Summary of Android	10
3.3.2	BLE Interface	10
3.3.3	Handling Data	11
3.3.4	Summary of Project, Structure and Classes	11
3.4	The End of the Pipeline: A Convenient Web Server	12
3.4.1	The Accelerometer Endpoint	13
3.4.2	The Speech Endpoint	14
3.4.3	User Interface	14
	References	16
	Appendices	17
	Setting up the Web Server	17
	Setting up the Android Application	17

List of Figures

1	GATT Profile Hierarchy	3
2	Architecture and Connectivity of the Overall System	5
3	FSM on the Discovery Board	7
4	Overview of the Android File Hierarchy	12
5	Screenshot of the web server's User Interface	15

List of Tables

1	JSON format for POST requests to the <code>/accelerometer</code> endpoint	13
2	Format for POST request files sent to the <code>/speech</code> endpoint	14
3	JSON format for the response returned by the speech endpoint to POST requests	14

1 Problem Statement

This text presents the design of a system whose ultimate goal is to gather data using the peripherals of a microcontroller and publish this data to a webserver via a chain of different communication media. From the user's perspective, the only setup required involves installing an Android application (designed for the purposes of this project) and launching a web server to collect the recorded data. After the Android device has paired with the Nucleo microcontroller, the data recorded from the accelerometer or microphone attached to the STM32F407 Discovery board will be transmitted automatically to the webserver, where the data may be visualized numerically and with pictorial support. If the data being sent is a microphone recording, the web server will perform *speech recognition* to parse a number n from the recorded audio, which will be communicated all the way back to the Discovery board, causing an LED to blink n times.

Given the reduced amount of resources usually present in microcontrollers, several devices were required to work in tandem in order to create a pipeline from the Discovery board to the Internet. The Discovery board, while having access to an accelerometer and an ADC for reading microphone data, has no resources for communicating over Bluetooth or HTTP. Instead, it may communicate via a wired connection to the Nucleo board that is fitted with a Bluetooth Low Energy (BLE) transmitter. Given that the Nucleo has no WiFi capabilities, it communicates data over BLE to an Android application, which may then tunnel the data over HTTP to the web server. Thus, in order to communicate sensor data from the Discovery board all the way to the web server, three distinct communication protocols are used (UART, BLE, HTTP), each having their own unique quirks and *impedimentum*. A major challenge of the design of this system revolved around the design of how to encapsulate data to be sent, and how to ensure the timing between transmitters and receivers was appropriate.

Furthermore, given the plethora of devices responsible for the transmission of data in this system, it was necessary to implement four distinct software components across three different technologies. As will be described further in this report, both the Discovery and Nucleo board required independent embedded-C programs. Furthermore, to bridge the gap between the microcontrollers and the web application, an Android app was devised and implemented in Java. Finally, the web application prompted the design of a RESTful API as well as a fancy interface for displaying the data it received. This was achieved mainly with the Python language and some of its excellent libraries.

Finally, another design challenge was that of speech recognition. Although a custom Convolutional Neural Network was initially designed for this task, the amount of time that would have been required to train it properly would have severely hindered progress on the rest of the project. Therefore, the speech recognition feature was outsourced to a Google Speech API.

The remainder of this text will describe how the system was designed, and will discuss the various challenges were experienced as well as how they were resolved. Ultimately, the system proposed in this paper is meant to show how embedded systems can communicate with each other, as well as how they may communicate over the vast Internet. Thus, the proposed system exhibits the *Internet of Things*, a field and concept that has been growing tremendously and is projected to grow into a market worth \$7.1 trillion by the year 2020[1].

The legendary philosopher John Milton once said “give me the liberty to know, to utter, and to argue freely according to conscience, above all liberties”[2]. The Internet of Things and the design proposed in this paper ultimately recognize microcontrollers as entities capable of knowing, uttering, and arguing, which was, above all else, demonstrated by the proposed system. The Internet of Things revolution, therefore, shall eventually bring the “Inter-computational Covenant on Civil and Political Rights”, allowing all devices to communicate equally to advance society. The authors hope that the results demonstrated in this text show promise in the evolution and advancements in the freedom of device-expression.

Each server device has only one GATT profile, but within the profile, it may have multiple services, which represent different features. For example, one service can be for a heart rate sensor, and another for a pressure sensor. In this project, there is only one custom made service. A service may have multiple characteristics, which represents different data. For example, one characteristic can be for a heart rate measurement and one can be for configuration settings for the heart rate sensor. Within each characteristic, there is a *Value*, the actual data. Then there contains one or many *Descriptors*, which describe a value. Finally, *Properties* contain the read, write and notify permissions for that characteristic values.

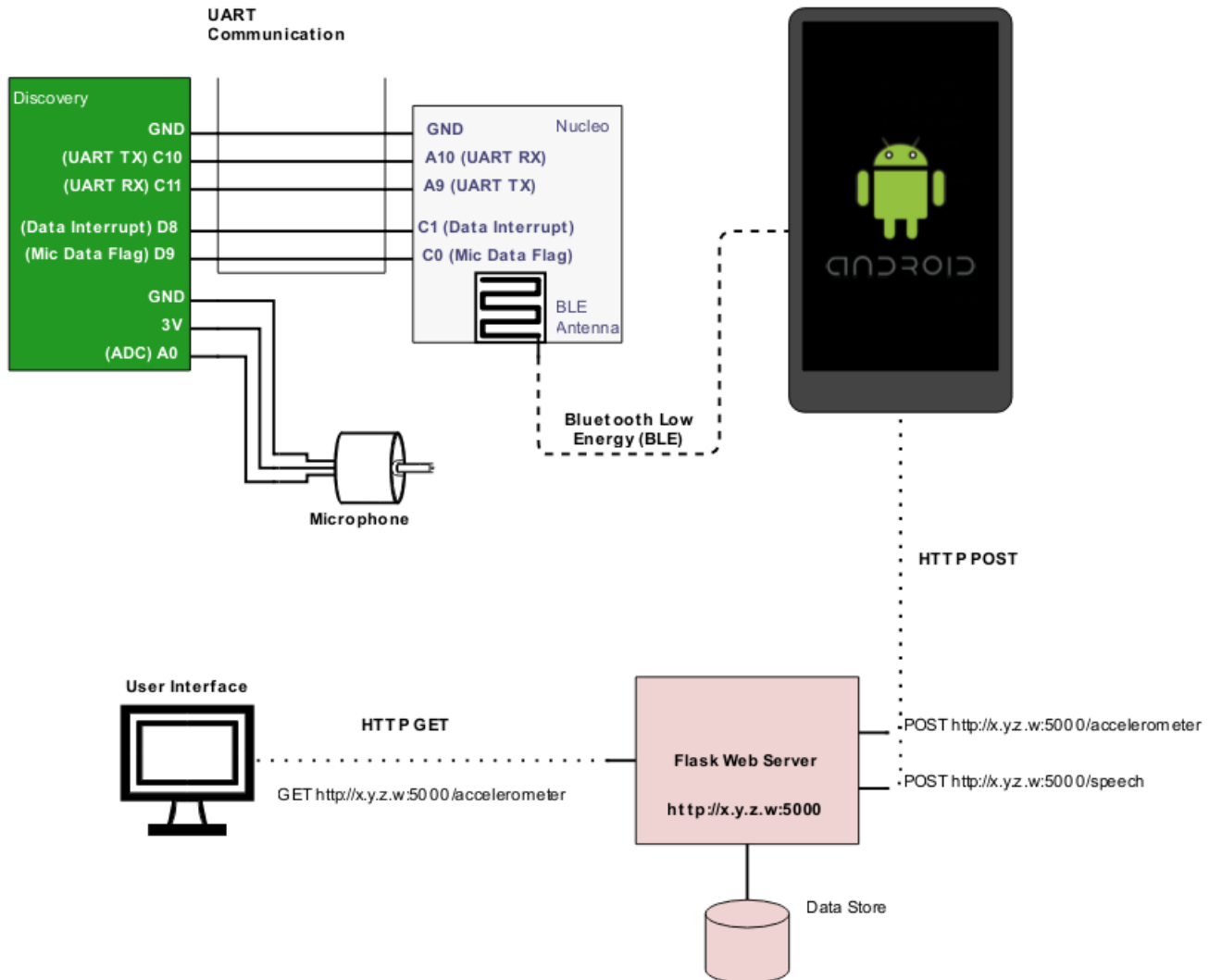
3 Implementation

3.1 An Overview of the System Architecture

Figure 2 shows the high-level architecture involving all parts of the system described in this paper. The pipeline begins at the Discovery board, where the user tapping the board prompts data to be recorded. This data is then transmitted over UART to the Nucleo board, which has the ability to transmit data via Bluetooth Low Energy. The Discovery and Nucleo blocks in Figure 2 explicitly demonstrate how to connect the two for proper UART communication.

The Android application discussed in this paper automatically connects to the Nucleo Board

Figure 2: Architecture and Connectivity of the Overall System



when the “SCAN FOR NUCLEO” button is pressed. Then, the app should receive the GATT ser-

vices and characteristics broadcast by the Nucleo, allowing for BLE communication to proceed.

With the Android app configured to send to the appropriate IP address where the Flask web server is hosted (that is, at `http://x.y.z.w:5000` in the example in Figure 2), it will automatically send the data it receives over BLE to the right endpoint of the web server. In the event that the Android app communicates microphone data, it will then wait for an HTTP response for the web server containing the perceived digit recognized from the audio samples, and the response will be sent backward through the pipeline until it eventually reaches the Discovery Board.

Upon receiving an HTTP POST at the `/accelerometer` endpoint, the web server will parse the accelerometer data from the body of the request, save the data in a CSV file on the server, and generate a graph of the data. Then, when the user interface is requested via HTTP GET, the graph and the CSV data are presented on a web page.

Upon receiving an HTTP POST at the `/speech` endpoint, the web server will extract the WAV file from the request and use Google Cloud’s Speech API to decipher a digit from the audio file. The digit that it perceives is then sent back to the sender of the audio data via an HTTP response.

When the digit response containing some integer n finds its way back to the Discovery board, the Discovery’s blue LED blinks n times. At this stage, the process starts over, where the Discovery waits for user input.

The remainder of this section discusses the implementation and functionality of all parts of the system in more detail.

3.2 The Beginning of the Pipeline: Handling the STM32 Discovery Board

The STM32F407 Discovery Board is the microcontroller that the user interacts with. As discussed previously, this board is responsible for gathering accelerometer and microphone data to send through the pipeline. Figure 3 demonstrates the Finite State Machine that the Discovery Board adheres to.

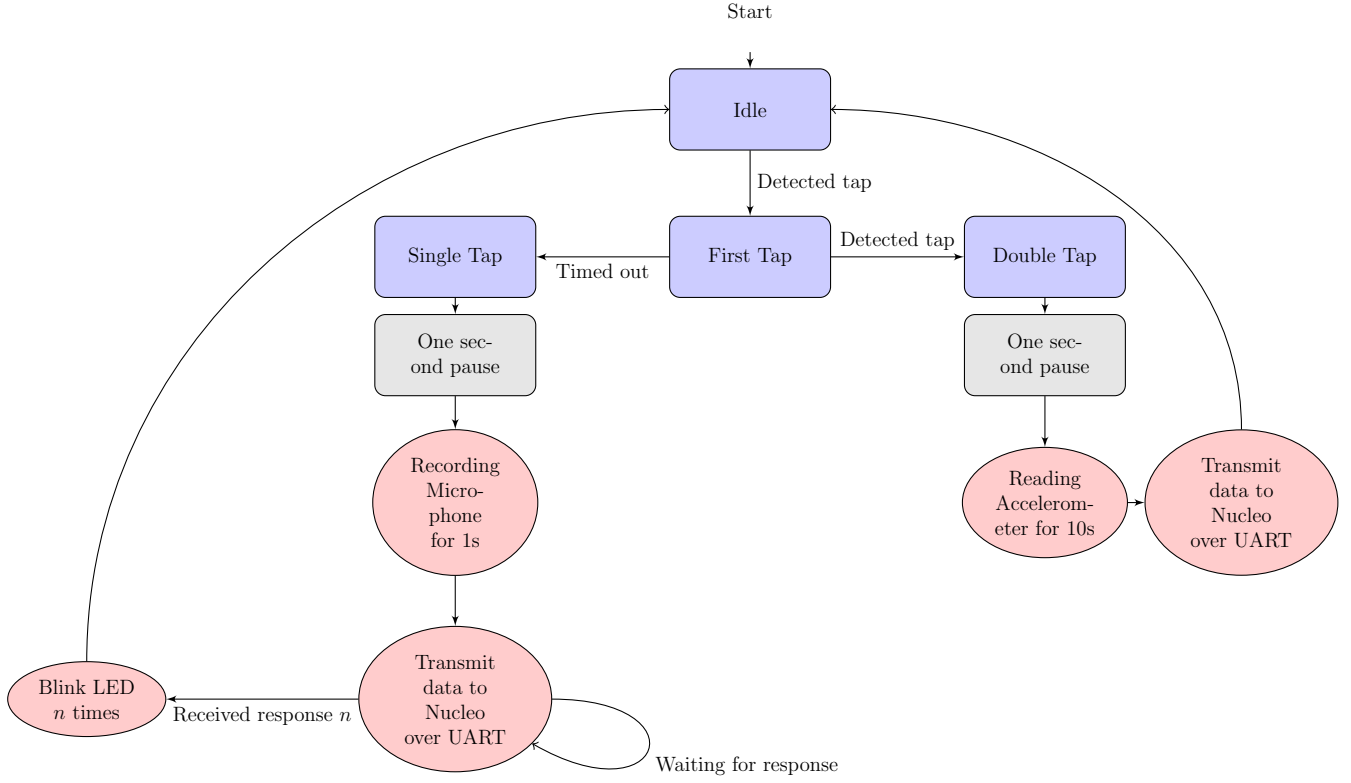
The system starts in the Idle state, where it waits for user input in the form of taps on the board itself. The taps are detected by accelerometer readings polled at 100Hz over 500ms intervals, when an accelerometer reading on the z axis surpasses the average reading over the window by a certain threshold. In order to filter out noisy accelerometer readings, an exponential moving average filter is applied, which filters as follows:

$$\hat{z}_t = \alpha z_t + (1 - \alpha) \hat{z}_{t-1}, \alpha \in (0, 1] \quad (1)$$

where z_t is the t th accelerometer reading, and \hat{z} is t th output of the filter. The parameter α was chosen through testing, and it represents how “smooth” the filter output should be.

When a tap is detected, the system enters the First Tap state, and the green LED is illuminated for visual feedback, so the user can gauge how rapidly to make a double tap. If no tap is detected in the next 500ms window, the system enters the Single Tap state (denoting that it has

Figure 3: FSM on the Discovery Board



detected only a single tap) and the blue LED is illuminated, and otherwise, it enters the Double Tap state where the orange LED is illuminated. These states ultimately determine which data to record, and how the rest of the system proceeds. In either case, there is a one second delay after the transition to these states to allow the user to prepare for the recording.

3.2.1 Sending Microphone Data

In the event that the system enters the Single Tap state, after the aforementioned one second grace period, the machine will begin to record data from the microphone. This data is sampled at 10kHz by the onboard analog to digital converter (ADC) for 1 second using direct memory access (DMA). During this recording period, the Discovery Board's red LED will be blinking. When the blinking stops, the recording has finished. In order to retain high audio quality, the ADC converts samples to 12-bit integers. Since the FSM is not in use during the recording period, it begins to wait on a signal before continuing. This effectively prohibits the FSM from being scheduled by the CPU until the signal is set. Once the DMA buffer is filled with ADC readings, a callback function is triggered, which in turn sets the signal allowing the FSM to resume.

After receiving the signal from the ADC DMA callback, the FSM can once again be scheduled. However, before sending the microphone data to the Nucleo board, it is desired to *compress* the

data returned by the DMA process. Since DMA buffers contain 32-bit words and ADC samples are only 12 bits wide, it is desirable to *squash* the data in the DMA buffer into 16-bit integers. This was done by an elegant **squash** algorithm, that performs this operation in place. This halves the amount of bytes required to send to the Nucleo board over UART.

With the compressed data ready, the Discovery Board asserts a **Data Interrupt** GPIO pin, which is directly connected to the Nucleo Board. This will trigger an interrupt on the Nucleo, essentially informing it that it should be ready to receive a UART transmission. Furthermore, it asserts a **Mic Data** GPIO pin, also connected directly to the Nucleo Board, which tells the Nucleo that it is about to receive microphone data. Then, the Discovery board begins transmitting the microphone data over UART.

Once the UART transmission is complete, the Discovery Board must wait for a response indicating how many times its LED should blink as indicated by the microphone reading. Since the amount of time necessary to receive the response is relatively enormous, it is undesirable for the Discovery Board to poll for UART responses. Thus, it initiates a UART Receive in Interrupt Mode, which will trigger an interrupt when the desired byte is received from the Nucleo Board. Meanwhile, the FSM waits on another signal in order to prevent it from being scheduled by the CPU. This signal is set when the UART Receive callback is triggered.

Upon receiving a response from the Nucleo Board, the discovery extracts a number $n \in \{0, 1, \dots, 9\}$ from its receive buffer. Then, the blue LED will blink n times, indicated the number that was recognized from the initial microphone reading. After the LED has finished blinking, the system returns to the Idle state, ready to start all over again. At this stage, no LED's should be illuminated.

3.2.2 Sending Accelerometer Data

In the event that the system enters the Double Tap state, after the aforementioned one second grace period, the machine will begin to record accelerometer readings at 100Hz for 10 seconds. As in the case of the microphone recording session, the red LED will blink for the duration of the accelerometer recording. In order to reduce the amount of data that will be sent over UART (and later, over BLE), the Discovery Board converts each accelerometer reading (x_i, y_i, z_i) into pairs (p_i, r_i) , where p_i and r_i are the corresponding pitch and roll measurements, respectively. Then, rather than sending $3 * 10000 = 30000$ floating point numbers, only $2 * 10000 = 20000$ floats must be sent.

Note that in contrast to the case of the microphone data, where it was desired to compress the readings into 16-bit integers, the accelerometer data should be kept at pairs of 32-bit floats to retain enough precision. However, before transmitting the data it is passed through an exponential moving average filter. However, this filter uses a less smooth α parameter as that which was used for detecting taps – this is due to the fact that it was easier to detect taps when the accelerometer readings were smoothed out as much as possible. To keep the integrity of the accelerometer readings, less filtering was done for the recorded data.

Once the Discovery Board has finished filtering all 10000 pairs, it is ready to transmit the data over UART to the Nucleo Board. As in the case with microphone data, the **Data Interrupt** pin

is asserted. However, in this scenario, the `Mic Data` GPIO pin is reset – this way, the Nucleo knows that it is about to receive accelerometer data. Next, the pitch and roll are sent in consecutive UART transmissions. The Discovery first transmits the buffer of pitch data, and waits 10ms before transmitting the roll data. The 10ms delay allows time for the Nucleo Board to prepare for another UART reception.

After transmitting the accelerometer data, it will eventually be uploaded to the Web Server implemented for this project. However, in the case of accelerometer data transmission, the Discovery Board *does not* get a response. Thus, it returns immediately to the Idle state, where the system is ready to start all over again. At this point, no LED's should be illuminated.

3.3 Smart Phone Android Application

Given that the Nucleo board has no WiFi/Internet capabilities, data must be communicated over BLE to an Android application, that then forwards the data to the web server over HTTP. For convention, this report may use "Android app" to signify the smart phone BLE mobile application. The report also assumes that the reader has basic knowledge of telecommunication protocols, and Android applications. This portion of the report details the implementation and the rationale behind the design of the smart phone application that acts as the intermediary component between the Nucleo board and the web server.

With that being said, the requirements for the Android application are the following:

- Scan for and connect to BLE peripheral devices;
- Enable the user the ability to start, or stop scanning for devices, and connect or disconnect from one peripheral device;
- Discover BLE services and characteristics from the peripheral device;
- Read voice and accelerometer data batches over BLE from the Nucleo board;
- Save the received data to its appropriate file;
- Once the accelerometer file contains 10 seconds worth of data, transmit the file to the web server over HTTP;
- Once the voice file contains 1 second worth of data, transmit the file to the web server over HTTP;
- Handle HTTP responses from the web server and send data to the Nucleo board.

Each of the Android application's business-logic features are implemented in the Java programming language, and the user interface is designed in XML. The implementation of the Android application is facilitated by using the Android Studio IDE.

3.3.1 Brief Summary of Android

The report doesn't go into detail about the code written for the project, if the reader desires to see the source code themselves, they can see it [here](#). There is documentation written within the code so that the reader may understand the basics of the code written. This report goes into detail about the high level designs and concepts used.

Android applications perform in a way such that all user interface and business-logic is performed within an *Activity*. Even if a developer doesn't need a user interface for their application, a main activity must instantiate and start once the user opens the application. Android developers must also declare any build dependencies such as programming libraries and hardware capabilities (such as Bluetooth and Internets) within the *manifest.xml* file. The activity, since there can be multiple activities, which runs on application start up is also declared in the manifest file.

3.3.2 BLE Interface

With regards to BLE, the smart phone is recognized as the *client device*, and the Nucleo board as the *server device*.

The first step for any BLE handshaking, is the *Scanning Phase*, which is facilitated by a *BLE Scan Callback*. For the client device to recognize BLE servers, the server must advertise itself and the client must scan for these advertisements. With the push of a *SCAN* button on the Android app user interface, the application enters a scanning mode which, by the help of a *Scan Callback* Java class, saves scanning results as MAC addresses that the Android application could try to connect to. Since scanning is cumbersome on the smart phone's battery, the scanning automatically stops after 10 seconds to save battery life. To facilitate full automation, the Android app automatically stops scanning once it finds the Nucleo board's MAC address and tries to connect to the board. This improves the time required to enable a connection between the client and server since the user doesn't have to look through a list of MAC addresses to find the Nucleo board's address and have to select that specific address.

The last phase for the BLE handsaking is the connection phase, which is facilitated by a *GATT Client Callback*, where GATT stands for *General Attribute*. Once the Android app finds the Nucleo board, it automatically tries to connect to it, the GATT Client Callback takes charge of this operation. On the press of a *CONNECT* toggle button, the user can decide to connect or disconnect from the selected peripheral device.

The GATT Client Callback handles any connection events, and outputs statuses such as *Connection Success* and *Connection Failure*. Since there isn't any authentication procedure between client and server devices, the GATT Client Callback simply notifies the server that it wishes to connect to it and once the server, by the help of a GATT Server Callback, receives the notification, and responds with a connection success. If somehow the connection is a failure, the Android application simply logs the result to the Android Studio debug console. Although, on a connection success, the Android application now starts to search for *Services* that the server may be transmitting. If the service used to package voice and accelerometer data is discovered, and contains their appropriate *Characteristics*, the Android application is ready to read and be notified by any

updates to the voice and accelerometer data.

At this point, the smart phone is connected to the Nucleo board over a BLE session and can now start to transmit data to one another! Exciting, if I do say so myself.

3.3.3 Handling Data

The Android app receives data in a byte array which is packaged under a characteristic assigned for either voice or accelerometer data. The batch of data is then written to its appropriate file with the help of the `AppController.java` class.

To send files over HTTP, the Android app makes use of the **Volley Library** to create and configure HTTP requests, while also handling any HTTP responses. This is useful if the developers wish to send files conveniently over HTTP as well as receive data from the web server over HTTP. The developers wished to keep data uploading simple. By encoding the file-to-be-sent into a base-64 string and adding the encoding to the HTTP request's form, the web server can easily retrieve the file from the request by reading and decoding the form body.

If the web server reads the voice data file, then the Android application receives a response containing the output from the Google Speech API, under the parameter "reading". This contains the number that the Google Speech API was able to identify given the voice data. The Android application sends this data to the Nucleo board over BLE by writing the value to its appropriate Characteristic. At this point, the data is now handled by the Nucleo server.

3.3.4 Summary of Project, Structure and Classes

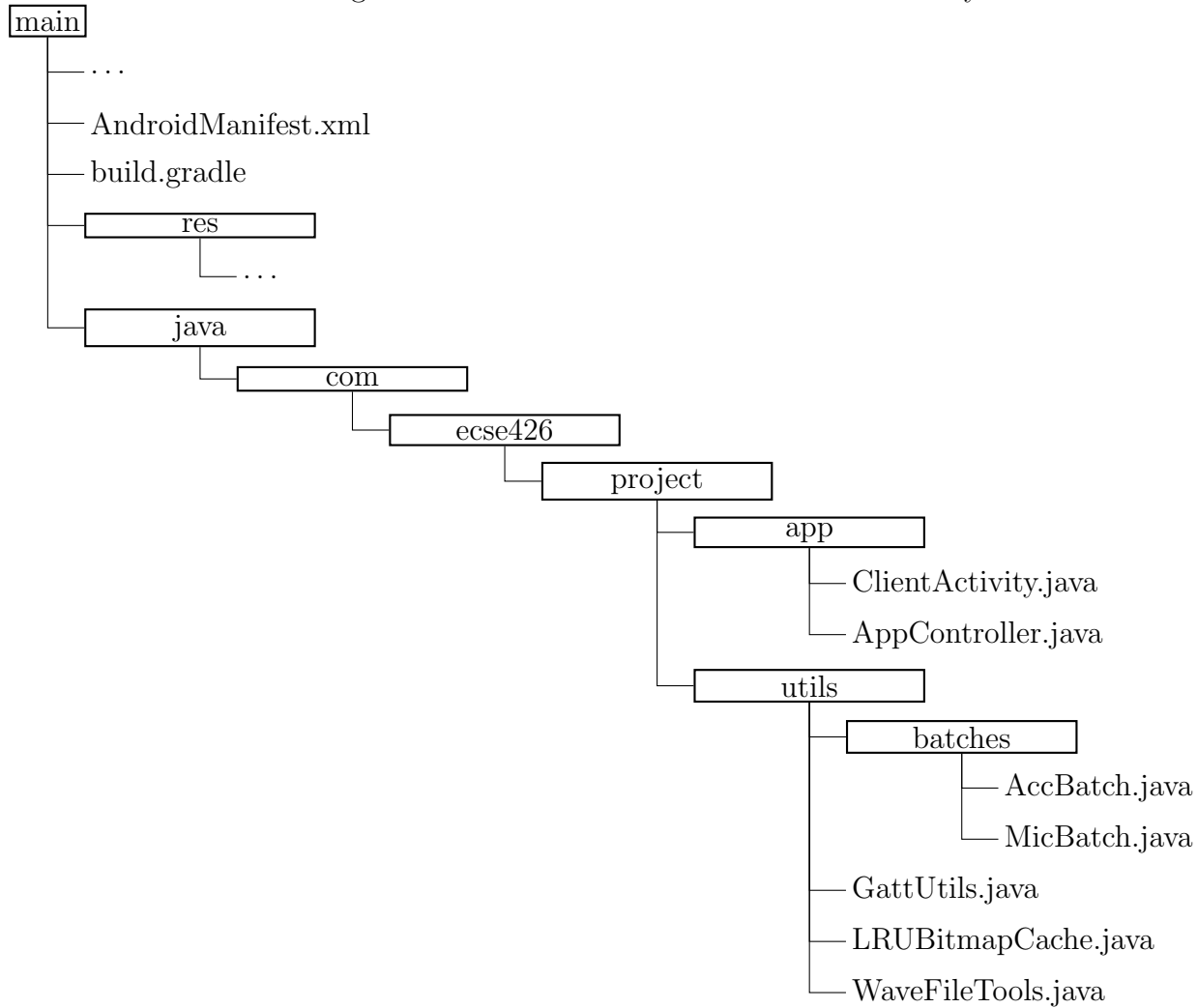
One can see a visualization of the Android app project structure in [Figure 4](#). Of course there are many files related to the application that couldn't be referenced under this report due to sizing constraints, any three dots represents that.

The `AndroidManifest.xml` is the manifest file specific to this application which contains dependencies. The `build.gradle` file is the gradle build script, Android Studio automatically runs its build configurations whenever the developer runs or builds the application.

The `ClientActivity.java` class is the main activity that runs on start up. It is the core of the application, every other class is written around it and is used to help it. This class contains the BLE Scan Callback and GATT Client Callback logic implemented under inner classes. It handles user input and contains function calls from other classes that receive and save data over BLE and HTTP.

The `AppController.java` handles HTTP requests by facilitating the Client Activity with a queue for requests. It also handles reading and writing voice and accelerometer data to their appropriate files. It is important to note that the `AppController.java` class extends the `Application` class, making it able to run simultaneously with the `ClientActivity.java` class and have data persist when the application is closed.

Figure 4: Overview of the Android File Hierarchy



The `AccBatch.java` and `MicBatch.java` classes are helper classes that contain functions to convert accelerometer or voice data respectively from bytes to a data type instructed by the function. The `GattUtils.java` class contains GATT Service, Characteristic, and Configuration UUIDs, as well as the Nucleo board's MAC address. The `WaveFileTools.java` class handles reading and writing data from and to files of Wave format. The `LRUBitmapCache.java` class is supposed to send and receive files, unfortunately it is not used.

Under the `res` directory, one can find all the user interface layouts and images. Any user interface design or image is found under that directory.

3.4 The End of the Pipeline: A Convenient Web Server

In order to perform more complex data processing and to have somewhere convenient to store the recorded data, the web server was implemented. This was achieved with the use of Python's Flask library for creating a minimal HTTP server with a REST API and a nice, simple interface to visualize the data that it receives.

The server has two main API endpoints, one for each of accelerometer and microphone data. Thus, if the server is running on 151.155.209.95:48080, the accelerometer endpoint is accessed via a `POST` request to 151.155.209.95:48080/accelerometer, and the microphone endpoint is accessed via a `POST` request to 151.155.209.95:48080/speech.

Flask manages API endpoints with a `Resource` interface. This interface provides `get()`, `post()`, and `put()` methods, which are called when the endpoint receives the corresponding HTTP packet. Thus, in the design of the web server, the `Accelerometer` and `Speech` classes were created that inherit from the `Resource` class.

3.4.1 The Accelerometer Endpoint

The `/accelerometer` endpoint accepts `POST` requests and is responsible for preprocessing accelerometer data, storing the data on the server, and plotting a graph of the data it receives. `POST` requests to this endpoint should have JSON bodies as described in Table 1. The API system is

Table 1: JSON format for `POST` requests to the `/accelerometer` endpoint

Parameter Name	Parameter Type	Description
"accelerometer"	[float]	List of accelerometer readings. Pitch values at even indices.
"starttime"	String	String indicating the time at which the recordings were sent to the API.

configured bind the `Accelerometer Resource` to the `/accelerometer` endpoint. As such, this `Resource` implements the `post()` method to handle the accelerometer data that it is sent via HTTP `POST`.

Since the "accelerometer" member of the `POST` request accepts an array of interleaved pitch and roll values, the data needs to be processed such that the pitch and roll could be stored in a more convenient format for plotting. The first task of this method is to parse the "accelerometer" member of the `POST` request to split up the pitch and roll data into their own respective lists. During this procedure, the data is written to the `data.csv` file on the server in comma-separated value (CSV) format. Each row of the CSV file contains a pitch reading and the corresponding roll reading. Furthermore, the "starttime" member of the `POST` is extracted and stored in a global variable so it can later be inscribed on the user interface.

With the pitch and roll values split into their own lists, they can be graphed with relative ease. The powerful Matplotlib[4] library is used to plot the graph, and the pitch and roll data are both plotted in the same plane. The plot is saved to the `static/graph.png` file, and is then available to be rendered on the user interface.

Since the accelerometer communication is a one-way procedure, the `Accelerometer Resource` has nothing interesting to return to the device that sent the `POST` request. Thus, its response contains only an acknowledgement string.

3.4.2 The Speech Endpoint

Similarly to the accelerometer endpoint, implementation of the speech functionality on the web server required the construction of a **Speech Resource**, which was bound to the `/speech` endpoint. The files portion of the body of the `POST` requests that this endpoint expects is shown in [Table 2](#).

With the WAV file extracted from the `POST` request, the task of the speech endpoint is to perform

Table 2: Format for `POST` request files sent to the `/speech` endpoint

Parameter Name	Parameter Type	Description
“audio”	WAV File	File containing audio recording in WAV format.

speech recognition. This is accomplished with the help of Google Cloud’s Speech API. Fortunately, Google Cloud provides a Python interface for their fabulous speech recognition algorithms. The API’s `transcribe_audio()` method is called on the WAV file, and a string is produced, containing the transcription of the audio recording. We try to cast this transcription to an integer, and if that fails we deduce that no number could be understood from the audio recording. In this scenario, we report a value of 0 in the `result` variable. Otherwise, `result` is assigned the number yielded by casting the transcription to `int`.

Contrarily to the case of the accelerometer endpoint, the microphone data communication process in this system is actually bidirectional. Thus, the sender of the microphone data expects a response from the web server, which will be passed all the way back to the Discovery board. The **Speech Resource**’s `post()` method returns a JSON object, which is illustrated in [Table 3](#).

Table 3: JSON format for the response returned by the speech endpoint to `POST` requests

Parameter Name	Parameter Type	Description
“reading”	<code>int</code>	Byte containing the number transcribed from the input audio file. If no number is detected, 0 is returned.

3.4.3 User Interface

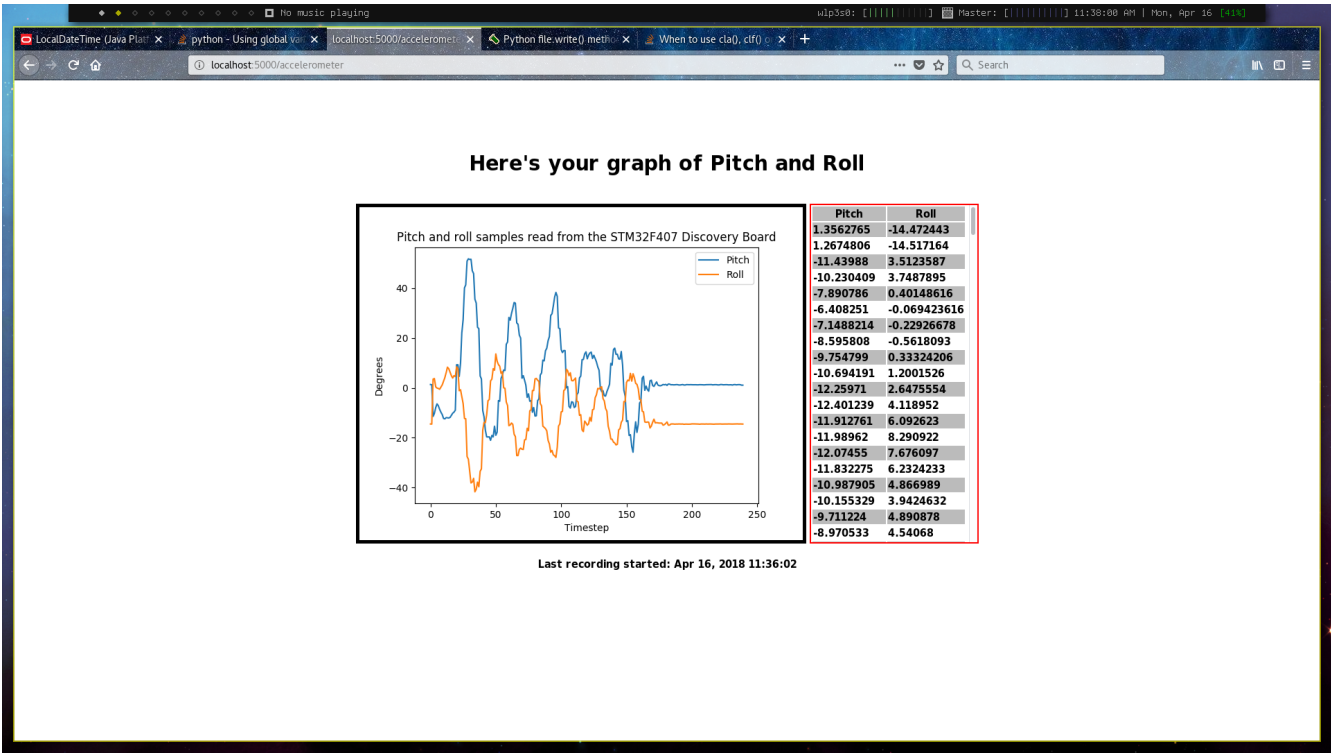
A user interface was implemented in order to have a convenient way of visualizing the accelerometer data. This was designed as a web page hosted by the Flask server described above. To realize this web page, a simple HTML layout was created that exploits Flask’s *template* functionality. Templates allow developers to dynamically pass parameters to HTML files – in this scenario, this was used to dynamically update the accelerometer data and recording time shown on the page.

To access the web page, whilst assuming the web server is running on `151.155.209.95:48080`, one would navigate to `151.155.209.95:48080/accelerometer` in their favorite web browser. The interface then displays the graph of the pitch and roll data, and shows the time at which the data was sent directly underneath the graph. Furthermore, the UI presents a scrolling table of the CSV

data saved in `data.csv` for the user to peruse comfortably. A screenshot of this user interface is shown in Figure 5.

In order to view the latest data, one must just refresh the web page. The graph displayed

Figure 5: Screenshot of the web server’s User Interface



on the user interface as well as the data used to generate the graph and the table shown are persisted on the server, so they can be viewed offline as well.

References

- [1] F. Wortmann and K. Flüchter, “Internet of things,” *Business & Information Systems Engineering*, vol. 57, no. 3, pp. 221–224, 2015.
- [2] K. Sanders, *Ethics and Journalism*. Sage, 2003.
- [3] “Gatt overview — bluetooth technology website.” <https://www.bluetooth.com/specifications/gatt/generic-attributes-overview>, 2018. Accessed: 2018-04-22.
- [4] J. D. Hunter, “Matplotlib: A 2d graphics environment,” *Computing In Science & Engineering*, vol. 9, no. 3, pp. 90–95, 2007.

Appendices

Appendix A - Setting up the Web Server

Some minor setup is necessary in order to run the web server, however the process is drastically simplified when taking advantage of some serendipitous Python tools. Firstly, Python 3 will be required. Furthermore, it is recommended to use `virtualenv` to keep dependencies managed. Finally, this tutorial will be assuming the user has access to the free open-source Python package manager, `pip`.

Firstly, navigate to the `api/` directory in this project's root. It is strongly recommended to create a `virtualenv` here, but it is not necessary. Next, execute the following command:

```
$ pip install -r requirements.txt
```

This will install all Python package dependencies required for the project. With these dependencies installed, the server is ready to run! Execute the following command, still within the `api/` directory:

```
$ python api.py
```

The command above launches the web server on port 5000. The server will run indefinitely. When it is desired to kill the server, navigate to the shell through which the server is being run and press `CTRL + C`.

Note that for the end-to-end pipeline described in this paper, the Android application must be configured to send HTTP packets to the IP of the machine running this server. In order to do this, navigate to the file at:

```
<project root>/android/app/src/main/java/com/ecse426/project/app/AppController.java
```

In this file, edit the string `WEBSITE_ADDRESS` to contain the IP address of the machine running the server. For the changes to take effect, the Android app must be rebuilt and reinstalled on an Android device.

Appendix B - Setting up the Android Application

Some setup is required to run the Android application since it is not available on the Google Play Store.

The user has to download and install the latest version of `Android Studio`. Once the IDE has been installed, the user must clone the source code from this `repository`.

Once the source code has been downloaded and the IDE installed, the user must `Import` the Android project by starting Android Studio and selecting the `Open existing Android project` within the Project Menu. From there, Android Studio prompts you with a File Explorer. Within the File Explorer, navigate to the source code that has just been downloaded and under `MicroP/project/android`, select the `build.gradle` file. Android Studio will import the project and automatically start downloading any dependencies required for the project.

Once that is complete, connect your Android phone via USB to your computer. Now within Android Studio, click on either `Run > Run` to install the application on your phone or on `Run > Debug` to install and launch a debug session for the application, allowing the user to view any logs within the console.

With either choice, the Android application is now installed and ready to use!